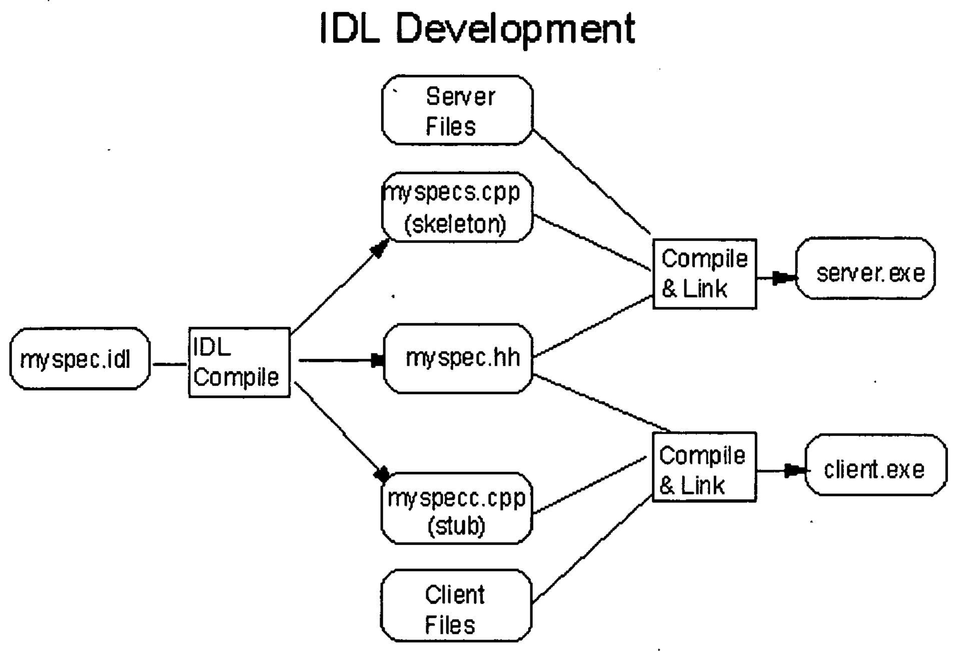
Interface Definition Language (IDL) 1Q

The Interface Definition Lnaguage (IDL) is technology-independent syntax for describing object encapsulations. It defines interfaces, thus creating objects. The IDL is only a definition language, mappings to programs languages are provided for invocation and manipulation. The interfaces have attributes and operation signatures. Each interface specified in the IDL, specifies the operation signatures through which requests are made. The IDL supports inheritance to promote reuse. Its specifications are compiled into header files and stub and skeleton programs.



An interface specification, myspec.idl is created using the Interface definition Language. The IDL input is then compiled (we are using a mapping to C++ as an example here), creating three files, myspecs.cpp,myspecc.cpp and myspec.hh. Myspecc.cpp and myspecs.cpp are implementation code for the declarations specified in myspecs.hh for the client and server respectively. Myspec.hh and myspecs.cpp are compiled with other server files to create the server or object implementation. Myspec.hh and myspecc.cpp are compiled with other client files to create the client. Myspecc.cpp is referred to as the stub and myspecs.cpp is referred to as the skeleton. Stub and skeleton programs are used to populate the Interface Repository. For the client, the stub program acts like a local function call. The stub performs encoding and decoding of parameters. The stub program is created within the native programming language.

For the object implementation (server) a skeleton program is created. The skeleton programs invoke the methods that are part of the object implementation. No implementation details are part of the 'skeleton', the developer adds the details. The IDL skeleton usually depends on an Object Adapter. Object adapters may create implementations 'on the fly', not making use of the skeleton program.

The Dynamic Invocation Interface allows dynamic creation of requests.

IDL Specifications

IDL specifications are module definitions, constant definitions, type definitions, interface or exception definitions. An IDL file may contain one or more specifications. Each specification forms a naming scope. A name can be used in an unqualified form within a particular scope. If it is unresolved within the scope, it is searched for in outer scopes. Names can also use the form :: to specifically resolve a name.

Names have global scope, thus the global qualified name
module1::interface1::enumeration1::indentifer1
would represent an identier1, defined within a enumeration (enumeratio1), defined within an interface (interface1) defined within a module (module1).

Interfaces are the main component in IDL specifications. They group attributes and operations. Interfaces are composed of a head and a body. The header specifies the header and any inheritance structure. Interfaces can inherit declarations from base interfaces. Multiple interface inheritance is supported. Ambiguities are resolved by qualifying identifiers.

Types in the IDL follow from the types defined in the CORBA Object Model.

Operations in the IDL follow from the operations defined in the CORBA Object Model.

Attributes of an interface are logically equivalent to a pair (set and get) of accessor functions. If an attribute is declared as read-only, no set function is defined.

Exception declarations take the form of structure definitions. They have identifier and member definitions. Standard runtime exceptions are defined for the ORB. Standard exceptions may be raised by any operation without an explicit raise clause. Each standard exception has an completion status (yes,no, maybe) which describes whether the method completed before the raising of the exception.

Dynamic Invocation Interface (DII)

The client can dynamically specify an object, operation and set of parameters. This information is usually retrieved from the Interface Repository. The client can name the type of object and operation to be performed. The client can build up call programmatically by issuing calls.

The client uses object references to issue requests. The semantics of the requests are identical to the IDL requests even though the process of building the request is different. The request has an operation name, object reference and parameters. The parameters are in a name-value list that is built at runtime. This list can handle a variable number of parameters so the DII can support any invocation.

Request are created using a Request pseudo-object (regular interface is provided but it may not be implemented as an object). Requests are first created and then sent out. A request is created by calling the create_request operation of the object to be invoked. The request interface has the following operations available: add_argument, invoke, delete, send and get_response. The parameter list is built up from successive calls to add_argument. The parameters to add_argument itself contain its name, data type, value, length and flags. Only value and length are required, the rest are used for type checking.

After created and built, the request can be invoked by using the invoke or send methods. The invoke method blocks the caller until the result comes back from the object implementation. The caller is not blocked when using a send operation. The get_response method can then be used to retrieve the response at a later time.

Repositories

Interface Repository (IR)

The Interface Repository maintains representations of the IDL definitions. It augments the DII by providing persistent objects which represent information about a servers interface. A client can query about an object unknown at compile time, query about services of the implementation and build a request.

The interfaces are maintained as Persistent 'live' objects available at runtime. The DII uses it to invoke requests of objects whose interfaces were unknown when the program was compiled. The IR is also used by the Object Request Broker for the marshaling and unmarshaling of parameters.

IR operations are defined for retrieving information objects maintained in the IR. The IR maintains information about interfaces and each interface is represented as an interface object. Each object in the IR can be one of the following types:

- AttributeDef - An object representing an attribute.
- ParameterDef - An object representing a parameter of an operation.
- ExceptionDef - An object representing an exception that may be raised.
- TypeDef - An object that represents a type definition.
- ConstantDef - An object representing a constant.
- OperationDef- An object representing an operation.
- InterfaceDef - Represents an interface definition. This object contains collection of object representing operations,types,attributes,constants and exceptions.
- ModuleDef - Collection of objects (interfaces,types, constants,exceptions) corresponding to a module definition.
- Repository - an object that maintains a collection of definitional objects.

Interfaces can be retrieved from the IR in three ways:

- Using the object reference, the InterfaceDef object can be retrieved. The collection of objects can then be traversed.
- Use the get_interface operation (part of the ORB interface).
- Traversing the IR structures using the names of the individual substructures.

Implementation Repository

- The Implementation Repository maintains information for the ORB to 'start-up' (and locate) an object implementation. The specification also envisions other information for an object such as debugging and versioning information. The makeup of the Implementation Repository is not specified in CORBA since it is platform independent.

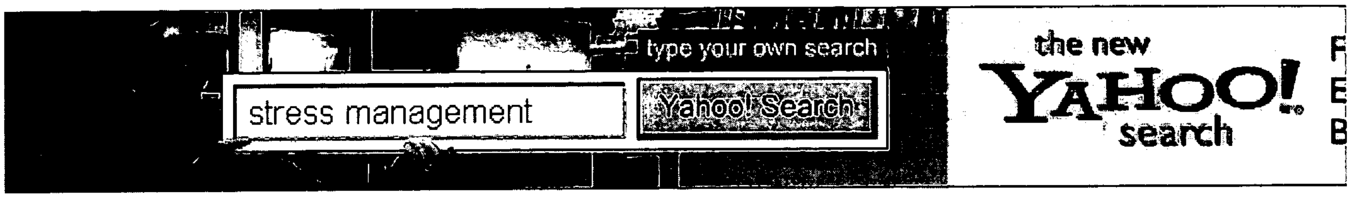
Dictionary.com

graph Look it up

[✓ Dictionary] Thesaurus

Home

Premium: Sign up | Login



ADVERTISEMENT | YOUR AD HERE

Dictionary - Thesaurus

1R

Get the Top 10 Most Popular Sites for "graph"

10 entries found for *graph*.

ADVERTISEMENT

graph[1] Pronunciation Key (grăf)
n.

1. A diagram that exhibits a relationship, often functional, between two sets of numbers as a set of points having coordinates determined by the relationship. Also called **plot**.
2. A pictorial device, such as a pie chart or bar graph, used to illustrate quantitative relationships. Also called **chart**.

tr.v. **graphed, graph·ing, graphs**

1. To represent by a graph.
2. To plot (a function) on a graph.

[Short for `graphic formula`.]

Source: *The American Heritage® Dictionary of the English Language, Fourth Edition*

[Buy it]

graph² [P] Pronunciation Key (grăf)
n.

1. The spelling of a word.
2. Any of the possible forms of a grapheme.
3. A written character that represents a vowel, consonant, syllable, word, or other expression and that cannot be further analyzed.

[Greek graphē, *writing*. See **graphic**.]

-graph
suff.

1. Something written or drawn: *monograph.*
2. An instrument for writing, drawing, or recording: *seismograph.*

[French -graphe, from Late Latin -graphus, from Greek -graphos, from graphein, *to write*. See gerbh- in Indo-European Roots.]

graph

[P] graph: log in for this definition of *graph* and other entries in *Merriam-Webster Medical Dictionary*, available only to Dictionary.com Premium members.

graph

[P] graph: log in for this definition of *graph* and other entries in *Merriam-Webster Medical Dictionary*, available only to Dictionary.com Premium members.

Source: *Merriam-Webster Medical Dictionary, © 2002 Merriam-Webster, Inc.*

graph

\Graph\, n. [See -graph.] (Math.) 1. A curve or surface, the locus of a point whose co["o]rdinates are the variables in the equation of the locus.

2. A diagram symbolizing a system of interrelations by spots, all distinguishable from one another and some connected by lines of the same kind.

Source: *Webster's Revised Unabridged Dictionary, © 1996, 1998 MICRA, Inc.*

graph

-graph \-graph\ (-gr[.a]f) [From Gr. gra`fein to write. See Graphic.] A suffix signifying something written, a writing; also, a writer; as autograph, crystograph, telegraph, photograph. Graphic \Graph"ic\ (gr[a^]f"[i^]k), Graphical\Graph"ic*al\ (-[i^]*kal), a. [L. graphicus, Gr. grafiko`s, fr. gra`fein to write; cf. F. graphique. See Graft.] 1. Of or pertaining to the arts of painting and drawing.

2. Of or pertaining to the art of writing.

3. Written or engraved; formed of letters or lines.

The finger of God hath left an inscription upon all his works, not graphical, or composed of letters. --Sir T. Browne.

4. Well delineated; clearly and vividly described.

5. Having the faculty of, or characterized by, clear and impressive description; vivid; as, a graphic writer.

Graphic algebra, a branch of algebra in which, the properties of equations are treated by the use of curves and straight lines.

Graphic arts, a name given to those fine arts which pertain to the representation on a fiat surface of natural objects; as distinguished from music, etc., and also from sculpture.

Graphic formula. (Chem.) See under Formula.

Graphic granite. See under Granite.

Graphic method, the method of scientific analysis or investigation, in which the relations or laws involved in tabular numbers are represented to the eye by means of curves or other figures; as the daily changes of weather by means of curves, the abscissas of which represent the hours of the day, and the ordinates the corresponding degrees of temperature.

Graphical statics (Math.), a branch of statics, in which the magnitude, direction, and position of forces are represented by straight lines

Graphic tellurium. See Sylvanite.>

Source: *Webster's Revised Unabridged Dictionary, © 1996, 1998 MICRA, Inc.*

graph

n : a drawing illustrating the relations between certain quantities plotted with reference to a set of axes [syn: graphical record] v 1: represent by means of a graph 2: plot upon a graph

Source: *WordNet ® 1.6, © 1997 Princeton University*

graph

1. <mathematics> A collection of nodes and edges.

See also connected graph, degree, directed graph, Moore bound, regular graph, tree.

2. <graphics> A visual representation of algebraic equations or data.

(1996-09-22)

Source: *The Free On-line Dictionary of Computing, © 1993-2003 Denis Howe*

graph

graph: in CancerWEB's On-line Medical Dictionary

Source: *On-line Medical Dictionary, © 1997-98 Academic Medical Publishing & CancerWEB*

Perform a new search, or try your search for "graph" at:

- Amazon.com - Shop for books, music and more
- AskJeeves.com - Get the top 10 most popular sites

- eLibrary - Search thousands of newspapers and magazines
- Google - Search the Web for relevant results
- Google Groups - Search Usenet messages back to 1981
- Merriam-Webster - Search for definitions
- Overture - Search the Web
- Roget's Thesaurus - Search for synonyms and antonyms

13

Dictionary - Thesaurus

Get the Top 10 Most Popular Sites for "protocol"

10 entries found for *protocol.*

pro·to·col Pronunciation Key (prōʹtə-kôlʹ, -kōlʹ, -kŏlʹ)
n.

a. The forms of ceremony and etiquette observed by diplomats and heads of state.
b. A code of correct conduct: *safety protocols; academic protocol.*
2. The first copy of a treaty or other such document before its ratification.
3. A preliminary draft or record of a transaction.
4. The plan for a course of medical treatment or for a scientific experiment.
5. Computer Science. A standard procedure for regulating data transmission between computers.

intr.v. **pro·to·coled,** or **pro·to·colled pro·to·col·ing,** or **pro·to·col·ling pro·to·cols** or **pro·to·cols**

To form or issue protocols.

[French protocole, from Old French prothocolle, *draft of a document*, from Medieval Latin prōtocollum, from Late Greek prōtokollon, *table of contents, first sheet* : Greek prōto-, *proto-* + Greek kollēma, *sheets of a papyrus glued together* (from kollān, *to glue together*, from kolla, *glue*).]

---

proʹto·colʹar (-kŏlʹər) or **proʹto·colʹa·ry** (-kŏlʹə-rē) *adj.*

protocol

[P] protocol: log in for this definition of *protocol* and other entries in *Merriam-Webster Dictionary of Law*, available only to Dictionary.com Premium members.

protocol

[P] protocol: log in for this definition of *protocol* and other entries in *Merriam-Webster Medical Dictionary*, available only to Dictionary.com Premium members.

protocol

\Pro"to*col\, n. [F. protocole, LL. protocollum, fr. Gr. ? the first leaf glued to the rolls of papyrus and the notarial documents, on which the date was written; prw^tos the first (see Proto-) + ? glue.] 1. The original copy of any writing, as of a deed, treaty, dispatch, or other instrument. --Burrill.

2. The minutes, or rough draught, of an instrument or transaction.

3. (Diplomacy) (a) A preliminary document upon the basis of which negotiations are carried on. (b) A convention not formally ratified. (c) An agreement of diplomatists indicating the results reached by them at a particular stage of a negotiation.

protocol

\Pro"to*col\, v. t. To make a protocol of.

protocol

\Pro"to*col\, v. i. To make or write protocols, or first draughts; to issue protocols. --Carlyle.

Source: *Webster's Revised Unabridged Dictionary, © 1996, 1998 MICRA, Inc.*

protocol

n 1: (computer science) rules determining the format and transmission of data [syn: communications protocol] 2: forms of ceremony and etiquette observed by diplomats and heads of state 3: code of correct conduct: "safety protocols"; "academic protocol"

Source: *WordNet ® 1.6, © 1997 Princeton University*

protocol

A set of formal rules describing how to transmit data, especially across a network. Low level protocols define the electrical and physical standards to be observed, bit- and byte-ordering and the transmission and error detection and correction of the bit stream. High level protocols deal with the data formatting, including the syntax of messages, the terminal to computer dialogue, character sets, sequencing of messages etc.

Many protocols are defined by RFCs or by OSI.

See also handshaking.

[Jargon File]

(1995-01-12)

Source: *The Free On-line Dictionary of Computing, © 1993-2003 Denis Howe*

protocol

n. As used by hackers, this never refers to niceties about the proper form for addressing letters to the Papal Nuncio or the order in which one should use the forks in a Russian-style place setting; hackers don't care about such things. It is used instead to describe any set of rules that allow different machines or pieces of software to coordinate with each other without ambiguity. So, for example, it does include niceties about the proper form for addressing packets on a network or the order in which one should use the forks in the Dining Philosophers Problem. It implies that there is some common message format and an accepted set of primitives or commands that all parties involved understand, and that transactions among them follow predictable logical sequences. See also handshaking, do protocol.

Source: *Jargon File 4.2.0*

protocol

protocol: in CancerWEB's On-line Medical Dictionary

Source: *On-line Medical Dictionary, © 1997-98 Academic Medical Publishing & CancerWEB*

Introduction on Intelligent Software Agents January 7 and January 9, 1997

2 Q

Agent based computing has been described as ``the next significant breakthrough in software development". The UK based consultancy firm Ovum has predicted that agent technology industry would grow from a US $37 million in 1994 to US $2.6 billion worldwide by the year 2000. Even though it is not exactly clear what is considered an agent and what not, this is a large number.

DEFINITIONS OF WHAT IS AN AGENT

What is an agent? (Part I)

" An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." (from Russell and Norvig, "Artificial Intelligence: a Modern Approach") The problem with this definition is that if we define the environment as whatever provides input and receives output, and we consider input to be sensing, and producing output to be acting then every program is an agent.

What is an agent? (Part II)

A weak notion (from Woodridge and Jennings) Essential properties of agents: * autonomy: agents operate without direct intervention of humans, and have control over their actions and internal state; * social ability: agents interact with other agents (and possibly humans) via an agent communication language; * reactivity: agents perceive their environment and respond in a timely and rational fashion to changes that occur in it; * pro-activeness: agents do not simply act in response to their environment, they are capable of taking the initiative (generate their own goals and act to achieve them).

What is an agent? (Part III)

A stronger notion (from Woodridge and Jennings) An agent has mental properties, such as knowledge, belief, intention, obligation. In addition, and agent has other properties such as: * mobility: agents can move around from one machine to another and across different system architectures and platforms; * veracity: agents do not knowingly communicate false information; * benevolence: agents always try to do what they are asked of; * rationality: agents will try to achieve their goals and not act in such a way to prevent their goals from being achieved.

What is an agent? (Part IV)

One last definition Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires (from IBM).

Are these agents?

* a thermostat
* a payroll program
* a spell checker

IS AGENT BASED PROGRAMMING A GOOD IDEA?

Agent based programming provides three abstractions:

1. the autonomous agent abstraction; 2. the cognitive agent abstraction; 3. the society of agents abstraction. Computer scientists use abstraction mechanisms (procedural abstractions, abstract data types, objects, etc.) all the time.

1. The autonomous agent abstraction
   * Autonomy implies ability to make decisions and to initiate actions without direct human supervision. This allows us to view agents as entities that have goals to achieve and are capable of initiating actions to achieve their goals.
   * Autonomy implies the existence of other autonomous entities. This allows us to see agents as members of a society of agents.

2. The cognitive agent abstraction
   * In everyday life, we use a folk psychology to explain and predict the behavior of complex intelligent systems (such as people). example: Maria intended to prepare her slides. example: Jane believed it was raining.
   * The intentional stance is a convenient way of talking about complex systems, which allows us to predict and explain their behavior without having to understand how their mechanism actually works.

3. The society of agents abstraction
   To be members of a society agents need:
   Shared transfer semantics: Agents need to be able to share data (for example, CORBA, OLE, etc).
   Shared domain semantics: Agents need to share the semantics of the domain on which they operate (for example, ontologies, process handbooks, STEP, etc.) An ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents.
   Shared task communication semantics: Agents need to share activities to accomplish their tasks, so they need representations of different types of interaction, agent communication languages, infrastucture for finding and routing, negotiation protocols, etc.
   Shared coordination semantics: Agents need to coordinate their activities, so they need models for task interactions, task dependencies, coordination, and negotiation.
   (from Charles Petrie)

WHAT'S SO SPECIAL ABOUT AGENT BASED COMPUTING

1. It can solve new problems
   Openness. When components of the system are not known in advance, change over time, and are highly heterogeneous (e.g. programming the Internet) an agent based approach allows to create systems that are flexible, robust, and can adapt to the environment by using their social skills, ability to negotiate with other agents, and ability to take advantage of opportunities.
   Complexity. With large and complex problems, agents offer a natural way to partition the problem into smaller and simpler components, that are easier to develop and maintain, and are specialized.

2. It can solve currently solved problems in a better way
   Distribution of data, control, resources. Many domains have distributed data, expertise, and resources. An agent based approach is more natural.
   Natural Metaphor. Agents provide an easy way to conceptualize metaphors. For instance, an e-mail filtering program can be presented using the metaphor of a personal digital assistant. This

could cause problems if the reasoning done by the agent is a black box to the user.
Legacy systems. Modifying existing software to keep pace with changing needs is often impossible. Legacy software can be incorporated into agent based software by building an ``agent wrapper" around it, to allow it to cooperate and communicate with other agent based software.

3. It requires new ways of thinking
   Non determinism. It is almost impossible to predict in advance how agents will interact. Agents can learn and change their behavior, it is not know what other agents will exist and what they will do.
   Global behavior emerges from local behavior. Individual behaviors can be designed for individual agents, but the global behavior of the system will emerge at run time. Minsky's ``Society of Mind" treats agents as individually very simple, but giving rise to ``intelligence" when acting together, ``in certain very special ways" in societies.
   The new agent society requires developing new social laws. Social laws need to be developed to make sure that agents deliver what they promise, do not harm other agents, and act rationally. Agents are supposed to carry out tasks on behalf of the user in an environment where the user has something significant on the line, such as money, reputation, privacy, security, etc. All the TV commercials where little dog icons set up meetings and buy opera tickets implicitly assume reliability and trust in addition to heuristic problem solving.

ARCHITECTURES FOR AGENTS

(from Russell and Norvig, "Artificial Intelligence: a Modern Approach", Chapter 2. Look there for more details and figures.)

Simple reflex agent

The agent works by finding a rule whose condition matches the current situation, as defined by perception, and then doing the action associated with the rule. The agent has no memory.

A reflex agent with internal state

The agent works by finding a rule whose condition matches the current situation, as defined by perception and by the stored internal state, and then doing the action associated with the rule. The internal state acts as a memory and allows for a better selection of the rule to apply.

An agent with explicit goals

The agent has an explicit goal, and when choosing an action it will select an action that achieves the goal. The decision process requires some plannig ("what will happen if I do action A"). This agent is more flexible than the simple reflex agent, and is capable of achieving goals.

A utility-based agent

The agent has a utility function that maps a state onto real number that describes how well the agent is performing. This agent not only achieves goals, but it maximizes some measure of performance.

HOW TO DESIGN AND IMPLEMENT AGENTS

There are two major approaches:

1. procedural approach: use a scripting language (such as Telescript or Tcl), or writing mobile code (such as Java applets);
2. declarative approach: define a universal agent communication language.

Procedural approaches

Use a scripting language (such as Tcl or Telescript) to write agents. A universal scripting language for all platforms and applications could replace APIs, formats, protocols.

Major issues:
* Portability: scripts must run everywhere. Higher level language is better.
* Connectivity: language must be embeddable. Must allow gluing and composition.
* Security: environment needs to be protected from agent (and viceversa)

Problems with Agents and the Web

* The Web is client/server oriented. Agents require peer-to-peer communication, because, to be agents, they need to be able to initiate messages to one another. Java applets can be used for agents, but need a router since they communicate only to the server that spawned them.
* The Web is oriented around formatting. The basic structure of documents is deigned to represent only transport and display of information. Agents require a structure that reflects task-level semantics.
(from Charles Petrie)

Declarative approaches

Software agents are software components that communicate with their peers by exchanging messages in an expressive agent communication language. The language must be sufficiently expressive to communicate all sorts of information, but also reasonably compact. Example: ARPA Knowledge Sharing Effort in which intercommunication between agents is performed using speech acts. This implements situated interactions more than ``intelligent" behavior in isolation.

WHERE TO FIND MORE INFORMATION

Special issue of Comm. of ACM, Vol 37, N 7, 1994.

M. Wooldridge and N. R. Jennings, "Intelligent agents: theories and practice", to appear in Knowledge Engineering Review, 10(2), 1995.

Jeffrey Rosenschein and Gilad Zlotkin, "Designing Conventions for Automated Negotiation" AI Magazine, Fall 1994, 29-46.

A good collection of information on Intelligent Software Agents is in http://www.sics.se/isl/abc/survey.html

An organized collection of information on agents is at http://www.cs.umbc.edu/agents/

OMG CORBA IDL

2R

Basic Concepts of CORBA's IDL

(obtained by Naji Ghazal, directly from GTE Labs' Distributed Object Computing Group)

The CORBA Specification, published by the Object Management Group, is both an architecture which encompasses an object model and a specification which defines among other things, an Interface Definition Language (IDL). IDL permits interfaces to objects to be defined independent of an objects implementation. After defining an interface in IDL, the interface definition is used as input to an IDL compiler which produces output that can be compiled and linked with an object implementation and its clients.

CORBA supports clients making requests to objects. The requests consist of an operation, a target object, zero or more parameters and an optional request context. A request causes a service to be performed on behalf of a client and results of executing the request returned to the client. If an abnormal condition occurs during execution of the request an exception is returned.

Interfaces can be used either statically or dynamically. An interface is statically bound to an object when the name of the object it is accessing is known at compile time. In this case, the IDL compiler generates the necessary output to compile and link to the object at compile time. In addition, clients that need to discover an object at run time and construct a request dynamically, can use the Dynamic Invocation Interface (DII). The DII is supported by an interface repository which is defined as part of CORBA. By accessing information in the interface repository, a client can retrieve all of the information necessary about an objects interface to construct and issue a request at run time. Since the static and dynamic requests are equally expressive semantically there will be no further distinction in this discussion between these two methods of invoking operations on objects.

CORBA has been postured as being both language-neutral and independent of ORB and CORBA object implementations. CORBA supports bindings for C but it is anticipated that it will eventually support language mappings for C++, Smalltalk and COBOL.

The source for the above account is:
The IDL Object Model, as part of ANSI X3H7 Object Model Features Matrix, at GTE Labs Distributed Object Computing Group.

The CORBA IDL Compiler for Java
Developed at Center for Computational Engineering at Sandia National Laboratories

Software Architecture Guidelines

25

Migrating to a CORBA environment provides for the opportunity for software architecture reengineering. Use the migration to CORBA as a means for improving your software architecture. An overview of software architecture development is given here.

Well designed software architectures (we view an architecture here as a collection of frameworks) are key in providing information systems with a long life. The software architecture is the 'glue' that remains in the system regardless of changes to external components. A good architecture will minimize cost as well as provide a stable environment. A stable system does not rely on specific component implementations. Cost will be minimized cost by reducing the complexity of component integration. This reduces programmer training time, development and maintenance costs. Develop the architecture interactively, learning from prototypes.

Make decisions based on market and standards awareness. Align your strategies with market and developed standards. Be wary of leading edge technologies. Having more than a few leading edge technologies can greatly increase your risk of failure. Choose a backup technology for each leading edge technology and develop your architecture so it can adapt to the backup technology. Expect product bugs and delays with leading edge technology and plan accordingly.

Depend on standards to reduce your risk. CORBA users (application developers) should write to the standard and not to specific non-standard product features. The goal is to insulate your application code so it is not effected as the product evolves.

Use the OMG/IDL for specifying the software architecture. The IDL allows for the specification of API's in a language and platform independent manner. The utilization of the IDL can be used independent of CORBA. In addition to ORB's IDL interfaces can be layered on top of Remote Procedure Call (RPC), DCE and library functions.

Software Architecture Development

A well designed software architecture provides: Interoperability between components and systems Extensibility Component interchangeability Cost savings through integration, operation and management savings.

We can think of software architecture as defining the boundaries between the major components of a software system These boundaries are interface specifications which can be expressed in the OMG/IDL. We strive to enhance system adaptability. A large portion of the cost of a software system, about 70 percent [Horowitz 93], are for operations and maintenance. The savings for a well structured vs. a poorly structured architecture typically exceed 50% [Horowitz 93]. CORBA is an enabler of software architecture and has simplified the creation of good software, providing architectural notations (OMG IDL) and system flexibility built into the ORB's. CORBA does not however design or define the software architecture.

Software architectures are designed by individual chief architects or by design teams. Individual architects can develop conceptually simple and elegant architectures. Once developed it is imperative that the architecture be completely understood by the individuals. Design teams usually create larger more complex architectures, the complexity a result of the inability to compromise and merge viewpoints. Once an architecture is designed, use prototypes and a regular release schedule for updates.

The OMG IDL and the ORB provide two basic sources for the enabling of software architectures. The IDL is a notation for specifying API's. These API's denote system level software boundaries, which is a goal of good software architecture. The OMG IDL provides a compilable linkage between software architecture and implementation. This allows the compiler to verify architectural constraints in the application software. The ORB supports flexibility and transparency in the implementation.

Design Process

In practice, formal design methodologies have had mixed results. The quality of the people on the team are more critical to success than the methodology. It should also be mentioned that some very competent people will have no concept of what makes a good architecture. Training is recommended for design staff. We will discuss the necessary insights to create an effective architecture.

The first step in the design process include farming and mining activities.

Farming involves the creation of abstractions of the system requirements. The system requirements are represented at the software architecture level. These requirements provide insight into the general operation of the system. It should be noted that the requirements are a snapshot of the users current requirements and while a good software architecture should support these requirements, the architecture must be flexible to handle anticipated and unanticipated future needs. Dependencies on commercial API's or business rules should be isolated to avoid obsolesces.

Mining involves the study of current technology and legacy systems. Models are developed of the existing subsystems. We wish to represent an idealized API for each component. With an understanding of the components, we generalize the components. It is important to study enough components to create a generalized class. To few, and the generalized class could be biased towards one implementation.

Once the background research has been completed, we begin the process of creating the software architecture. This is the 'artistic' step of the process. A designer may start with a strawman to be further refined, or some other method based on experience. Design reviews can play an important role in the design process. This also helps to educate other members as in the structure of the software architecture.

Once the architecture has been stabilized a prototype should be developed. Once the prototype is complete, the changes should be considered very carefully. The updates should be released as discrete updates.

Wrapper technology

Object wrapping is a practice for implementing a software architecture given pre-existing components. A wrapper provides a more controllable form of a legacy system that is more suitable for integration with other components. Wrappers provide technology migration paths for legacy systems, allowing a component upgrade without affecting the rest of the system.

The wrapper provides a clean interface to the legacy system. Before object wrapping a legacy system has: unique API Unique access mechanism Nonexchangeable data format Limited metadata Inadequate interoperability Limited/proprietary security After wrapping a system should have The desired API Uniform Access mechanism Exchangeable data format Complete/uniform metadata Meets interoperability needs uniform security interfaces.

Object wrapping approaches can be divided into the following categories: Layering - Layering is a mapping from one form of API to another. An example of this would be the layering of a CORBA-based interface over some RPC services. Layering can be used to aggregate existing components or partition complex systems. Data Migration - Many legacy systems are actually very large database applications. Anomalies can exist in these systems that can be handled by migrating the data to a new data model or by developing a wrapper to provide access to the database application. Reengineering Applications - Once an application is wrapped, it can be reengineered to provide increased performance or enhanced maintainability. Middleware - Middleware is a general term used to describe a large array of system integration software. Two major categories of middleware are distributed processing middleware (lower middleware) and database middleware (upper middleware). Commercial products that support CORBA are available in both categories. Encapsulation - An encapsulation is a black-box abstraction that hides the underlying implementation by means of an interface. It completely separates the interface from implementation. CORBA encapsulations hide differences in programming languages, location, operating system and data structure. Wrappers for architecture Implementation - The wrapper provides interoperability between the architecture and legacy systems. These wrappers could provide added value, such as metadata for the legacy system. Metadata is used by applications to discover information about object implementations. Wrappers for mediators and brokers - To support a diverse array of processing services, brokers and mediators are required. Brokers and mediators support such functions as access to disparate information sources, conversion of incompatible data formats and sophisticated search and data presentation algorithms.

ORB Selection

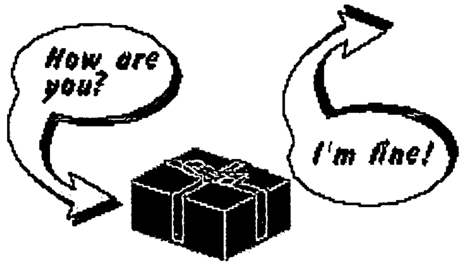
Choosing an ORB requires a thorough understanding of an applications present and future software requirements as well as how candidate ORB solutions satisfy these needs. This analysis is usually performed at the software architecture level. After development of this understanding, an informed decision can be made. The following guidelines are based on SAAM (Software Architecture Analysis Method).

This method involves the following five steps:

1. Develop scenarios of the anticipated or desired uses of the system. Consider various roles (user, developer, system administrator) involved in the target system. Scenarios fall into two categories, direct scenarios and indirect scenarios. Direct scenarios are those that can be supported without changing the architecture. Indirect scenarios require the changing of 1 or more components of the architecture.
2. Describe the candidate architectures, capturing the data components and connections. Describe in a notation that is acceptable to all parties.
3. Perform scenario evaluation. Record the needed changes for each indirect scenario. Develop a cost associated with these changes.
4. Determine scenario interaction - Record any indirect scenarios that affect the same components. Scenario interaction measures the separation of concerns, assuming each scenario is inherently different. The architecture with the fewest scenario interactions would be favored.
5. Ranking - Apply a weighting or ranking scheme to the scenarios and scenario interactions.

WebCrawler SELECT

3Q



What is Object-Oriented Software?

by Terry Montlick

Software Design Consultants

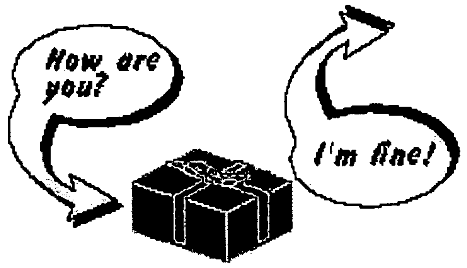
The Heart of the Matter - The "Black Box"

Object-oriented software is all about **objects**. An object is a **"black box"** which receives and sends *messages*. A black box actually contains *code* (sequences of computer instructions) and *data* (information which the instructions operates on). Traditionally, code and data have been kept apart. For example, in the C language, units of code are called *functions*, while units of data are called *structures*. Functions and structures are not formally connected in C. A C function can operate on more than one type of structure, and more than one function can operate on the same structure.

Not so for object-oriented software! In *o-o* (object-oriented) programming, code and data are merged into a single indivisible thing -- an object. This has some big advantages, as you'll see in a moment. But first, here is why SDC developed the "black box" metaphor for an object. A primary rule of object-oriented programming is this: as the user of an object, **you should never need to peek inside the box!**

An object

Why shouldn't you need to look inside an object? For one thing, all communication to it is done via **messages**. The object which a message is sent to is called the **receiver** of the message. Messages define the *interface* to the object. Everything an object can do is represented by its message interface. So you shouldn't have to know anything about what is in the black box in order to use it.

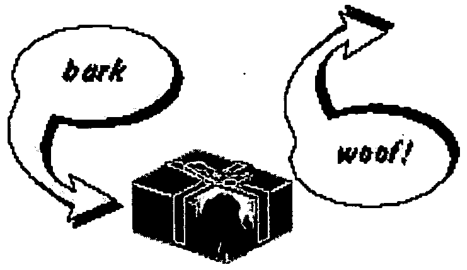


And not looking inside the object's black box doesn't tempt you to directly modify that object. If you did, you would be tampering with the details of how the object works. Suppose the person who programmed the object in the first place decided later on to change some of these details? Then you would be in trouble. Your software would no longer work correctly! But so long as you just deal with objects as black boxes via their messages, the software is guaranteed to work. Providing access to an object only through its messages, while keeping the details private is called *information hiding*. An equivalent buzzword is *encapsulation*.

Why all this concern for being able to change software? Because experience has taught us that software changes. A popular adage is that *"software is not written, it is re-written"*. And some of the costliest mistakes in computer history have come from software that breaks when someone tries to change it.

Classes

How are objects defined? An object is defined via its **class**, which determines everything about an object. Objects are individual *instances* of a class. For example, you may create an object call Spot from class *Dog*. The Dog class defines what it is to be a Dog object, and all the "dog-related" messages a Dog object can act upon. All object-oriented languages have some means, usually called a *factory*, to "manufacture" object instances from a class definition.



Spot

You can make more than one object of this class, and call them Spot, Fido, Rover, etc. The Dog class defines messages that the Dog objects understand, such as "bark", "fetch", and "roll-over".

You may also hear the term **method** used. A method is simply the action that a message carries out. It is the code, which gets executed when the message is sent to a particular object.

Arguments are often supplied as part of a message. For example, the "fetch" message might contain an argument that says *what to fetch*, like "the-stick". Or the "roll-over" message could contain one argument to say *how fast*, and a second argument to say *how many times*.

Some Real Life Examples

If you wanted to add two numbers, say, 1 and 2, in an ordinary, non-object-oriented computer language like C (don't worry -- you don't need to know any C to follow this), you might write this:

```
a = 1;
b = 2;
c = a + b;
```

This says,

> "Take *a*, which has the value 1, and *b*, which has the value 2, and add them together using the C language's built-in addition capability. Take the result, 3, and place it into the variable called *c*."

Now, here's the same thing expressed in Smalltalk, which is a pure object-oriented language:

```
a := 1.
b := 2.
c := a + b.
```

Wait a minute. Except for some minor notational differences, this looks exactly the same! Okay, it *is* the same, but the meaning is dramatically different.

In Smalltalk, this says,

> "Take the object *a*, which has the value 1, and send it the message *"+"*, which included the argument *b*, which, in turn, has the value 2. Object *a*, receive this message and perform the action requested, which is to add the value of the argument to yourself. Create a new object, give this the result, 3, and assign this object to *c*."

Hmm. This seems like a far more complicated way of accomplishing exactly the same thing! So why bother?

The reason is that objects greatly simplify matters when the data get more complex. Suppose you wanted a data type called list, which is a list of names. In C, list would be defined as a *structure*.

```
struct list {
<definition of list structure data here>
};

list a, b, c;

a = "John Jones";
b = "Suzy Smith";
```

Let's try to add these new a and b in the C language:

```
c = a + b;
```

Guess what? This doesn't work. The C compiler will generate an error when it tries to compile this because it doesn't know what to do with *a* and *b*. C compilers just know how to add numbers. Period. But *a* and *b* are not numbers. One can do the same thing in Smalltalk, but this time, list is made a *class*, which is a subclass of the built-in Smalltalk class called "String":

```
a := List fromString: 'John Jones'.
b := List fromString: 'Suzy Smith'.
c := a + b.
```

The first two lines simply create List objects *a* and *b* from the given strings. This now works, because the list class was created with a method which specifically "knows" how to handle the message "+". For example, it might simply combine the argument with its own object by sticking them together with a comma separating them (this is done with a single line of Smalltalk). So *c* will have the new value:

```
'John Jones, Suzy Smith'
```

Here's another example of a "+" message using more interesting objects. Click a, b, and c in turn to find out what they are (you'll need a WWW browser with support for "aiff" sounds):

c = a + b;

Using Non-Object-Oriented Languages

It's also possible to use objects and messages in plain old non-object-oriented languages. This is done via function calls, which look ordinary, but which have object-oriented machinery behind them. Among other things, this allows sophisticated client-server software to run "transparently" from within ordinary programming languages.

Suppose you added a "plus" function to a C program:

```
int plus(int arg1, int arg2)
{return (arg1 + arg2); }
```

This hasn't really bought you anything yet. But suppose that instead of doing the addition on your own computer, you automatically sent it to a server computer to be performed:

```
int plus(int arg1, int arg2)
{ return server_plus(arg1, arg2); }
```

The function server_plus() in turn creates a message containing *arg1* and *arg2*, and sends this message, via a network, to a special object which sits on a server computer. This object executes the "plus" function and sends the result back to you. It's object-oriented computing via a back-door approach!

This example is not very fancy, and, of course, it's easier to simply add two numbers directly. But as the musical example illustrated, there's no limit to the complexity of an object. A single object can include entire databases, with millions of pieces of information. In fact, such database objects are common in client-server software.

This also illustrates the flexibility of the object-oriented approach. In the usage just described, the object is very different from the earlier "a + b" example. Here, it receives *two* arguments, namely, the two objects that it is supposed to add. Previously, in the Smalltalk example, the object that was receiving a message was the first object, *a*. But in a client-server environment, the addition is not done locally, on the client machine, but remotely, on a server machine. The server machine contains the object that the message is sent to, and since it doesn't know anything about the first argument, you have to send *both* arguments.

Inheritance

If there is already a class which can respond to a bunch of different messages, what if you wanted to make a new, similar class which adds just a couple of more messages? Why have to re-write the entire class?

Of course, in any good object-oriented language, you don't. All you need to do is create a **subclass** (or **derived class**, in C++ terminology) of the original class. This new class **inherits** all the existing messages, and therefore, all the behavior of the original class. The original class is called the **parent class,** or **superclass**, of the new class. Some more jargon -- a subclass is said to be a **specialization** of its superclass, and the conversely a superclass a **generalization** of its subclasses.

Inheritance also promotes **reuse**. You don't have to start from scratch when you write a new program. You can simply reuse an existing repertoire of classes that have behaviors similar to what you need in the new program.

For example, after creating the class *Dog*, you might make a subclass called *Wolf*, which defines some wolf-specific messages, such as *hunt*. Or it might make more sense to define a common class called *Canis*, of which both Dog and Wolf are subclasses.

Much of the art of o-o programming is determining the best way to divide a program into an economical set of classes. In addition to speeding development time, proper class construction and reuse results in far fewer lines of code, which translates to less bugs and lower maintenance costs.

Object-Oriented Languages

There are almost two dozen major object-oriented programming languages in use today. But the leading *commercial* o-o languages are far fewer in number. These are:

- **C++**
- **Smalltalk**
- **Java**

C++

C++ is an object-oriented version of C. It is compatible with C (it is actually a superset), so that existing C code can be incorporated into C++ programs. C++ programs are fast and efficient, qualities which helped make C an extremely popular programming language. It sacrifices some flexibility in order to remain efficient, however. C++ uses *compile-time binding,* which means that the programmer must specify the specific class of an object, or at the very least, the most general class that an object can belong to. This makes for high run-time efficiency and small code size, but it trades off some of the power to reuse classes.

C++ has become so popular that most new C compilers are actually C/C++ compilers. However, to take full advantage of object-oriented programming, one must program (and think!) in C++, not C. This can often be a major problem for experienced C programmers. Many programmers think they are coding in C++, but instead are only using a small part of the language's object-oriented power.

Smalltalk

Smalltalk is a pure object-oriented language. While C++ makes some practical compromises to ensure fast execution and small code size, Smalltalk makes none. It uses *run-time binding,* which means that nothing about the type of an object need be known before a Smalltalk program is run.

Smalltalk programs are considered by most to be significantly faster to develop than C++ programs. A rich class library that can be easily reused via inheritance is one reason for this. Another reason is Smalltalk's dynamic development environment. It is not explicitly compiled, like C++. This makes the development process more fluid, so that "what if" scenarios can be easily tried out, and classes definitions easily refined. But being purely object-oriented, programmers cannot simply put their toes in the o-o waters, as with C++. For this reason, Smalltalk generally takes longer to master than C++. But most of this time is actually spent learning object-oriented methodology and techniques, rather than details of a particular programming language. In fact, Smalltalk is syntactically very simple, much more so than either C or C++.

Unlike C++, which has become standardized, The Smalltalk language differs somewhat from one implementation to another. The most popular commercial "dialects" of Smalltalk are:

- **VisualWorks** from *ParcPlace-Digitalk, Inc.*
- **Smalltalk/V and Visual Smalltalk** from *ParcPlace-Digitalk Inc.*
- **VisualAge** from *IBM*

VisualWorks

VisualWorks is arguably the most powerful of Smalltalks. VisualWorks was developed by ParcPlace, which grew out of the original Xerox PARC project that invented the Smalltalk language. VisualWorks is *platform-independent,* so that an application written under one operating system, say, Microsoft Windows, can work without any modification on any of a wide range of platform supported by ParcPlace, from Sun Solaris to Macintosh. VisualWorks also features a *GUI* (Graphic User Interface) builder that is well-integrated into the product.

Smalltalk/V and Visual Smalltalk

Digitalk's versions of Smalltalk are somewhat smaller and simpler, and are specifically tailored to IBM compatible PCs. A Macintosh version was available, but support has since been abandoned. This does not bode well for Digitalk cross-platform efforts. Digitalk has a separate GUI builder, called PARTS Workbench (bundled with Visual Smalltalk), which allows quick construct of an application.

ParcPlace and Digitalk were merged into a single company, ParcPlace-Digitalk, Inc. The future of the Digitalk product line is uncertain, and it may just be spun off back into a separate company.

VisualAge

IBM's version of Smalltalk, VisualAge, is comparable to Smalltalk/V with PARTS. Both of these Smalltalks allow programmers to readily exploit machine-specific features, at the expense of some portability. IBM has adapted existing industry standards for such things as file management and screen graphics. When IBM talks, people listen, and IBM has made a substantial commitment to Smalltalk.

Java

Java is the latest, flashiest object-oriented language. It has taken the software world by storm due to its close ties with the Internet and Web browsers. It is designed as a portable language that can run on any web-enabled computer via that computer's Web browser. As such, it offers great promise as the standard Internet and Intranet programming language.

Java is a curious mixture of C++ and Smalltalk. It has the syntax of C++, making it easy (or difficult) to learn, depending on your experience. But it has improved on C++ in some important areas. For one thing, it has no *pointers*, low-level programming constructs that make for error-prone programs. Like Smalltalk, it has *garbage collection*, a feature that frees the programmer from explicitly allocating and de-allocating memory. And it runs on a Smalltalk-style *virtual machine*, software built into your web browser which executes the same standard compiled Java *bytecodes* no matter what type of computer you have.

Java development tools are being rapidly deployed, and are available from such major software companies as IBM, Microsoft, and Symantec.

In Summary

Object-oriented programming offers a new and powerful model for writing computer software. Objects are "black boxes" which send and receive messages. This approach speeds the development of new programs, and, if properly used, improves the maintenance, reusability, and modifiability of software.

O-o programming requires a major shift in thinking by programmers, however. The C++ language offers an easier transition via C, but it still requires an o-o design approach in order to make proper use of this technology. Smalltalk offers a pure o-o environment, with more rapid development time and greater flexibility and power. Java promises much for Web-enabling o-o programs.

Go to SDC Home Page